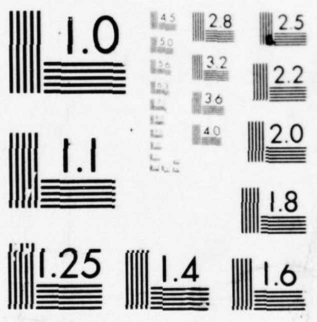
AD-A038 462 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
EVALUATION OF ALTERNATIVE COMPUTER ARCHITECTURES.(U)
FEB 77 M R BARBACCI, W E BURR, S H FULLER F44620-73-C-0074
UNCLASSIFIED AFOSR-TR-77-0381 NL

2 OF 2
AD
A038462

END
DATE
FILMED
5-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

INDIVIDUAL M MEASURE

Test Program	Computer Architecture		
	IBM S/370	PDP-11	Interdata 8/32
A. Priority I/O Kernel	212 [3]	28 [4]	28 [12]
	354 [12]	24 [12]	32 [14]
	522 [14]	24 [14]	28 [17]
B. FIFO I/O Kernel	424 [2]	208 [2]	192 [2]
	920 [13]	188 [3]	226 [4]
	434 [17]	296 [13]	114 [13]
C. I/O Device Handler	328 [1]	309 [1]	426 [1]
	304 [17]	290 [17]	279 [17]
D. Large FFT	10810 [11]	14746 [11]	10886 [11]
	10810 [9]*	14746 [9]*	8560 [9]*
			8560 [17]A
E. Character Search	854 [1]	730 [1]	958 [1]
	940 [4]	770 [11]	1044 [3]
	1724 [11]	520 [17]	1021 [11]
F. Bit Test, Set, Reset	378 [9]	162 [3]	222 [4]
	358 [12]	178 [9]	176 [9]
	238 [17]	152 [12]	296 [11]A
			276 [12]
G. Runge-Kutta Int.	141074 [2]	102662 [2]	100062 [2]
	228056 [17]	94960 [3]	100042 [4]
		176960 [17]	117984 [11]A
			138414 [17]
H. Linked List Insertion	228 [4]	204 [13]	224 [3]
	304 [13]	218 [14]	260 [13]
	264 [14]	240 [17]	238 [14]
I. Quicksort	1024 [5]	14960 [5]	2968 [5]
	1008 [6]	2756 [6]	1732 [6]
J. ASCII to Float-Pt.	241 [4]	292 [5]	363 [3]
	437 [5]	275 [7]	423 [5]
	433 [7]	283 [17]	334 [7]
K. Boolean Matrix	832 [3]	582 [4]	384 [6]
	909 [6]	776 [6]	566 [8]
	896 [8]	932 [8]	640 [17]
L. Virtual Memory Exchange	532 [3]	541 [4]	721 [7]
	532 [7]	566 [7]	1058 [8]
	645 [8]	945 [8]	780 [17]

INDIVIDUAL R MEASURES

Test Program	Computer Architecture		
	IBM S/370	PDP-11	Interdata 8/32
A. Priority I/O Kernel	947 [3]	108 [4]	166 [12]
	2146 [12]	106 [12]	166 [17]
	3052 [14]	106 [14]	214 [14]
B. FIFO I/O Kernel	2222 [2]	1096 [2]	698 [2]
	4583 [13]	810 [3]	937 [4]
	2226 [17]	1419 [13]	482 [13]
C. I/O Device Handler	1789 [1]	1480 [1]	1902 [1]
	1729 [17]	1416 [17]	1391 [17]
D. Large FFT	62904 [11]	70512 [11]	60446 [11]
	62904 [9]*	70512 [9]*	50045 [9]*
			50045 [17]A
E. Character Search	5603 [1]	4348 [1]	5885 [1]
	5549 [4]	4326 [11]	3139 [3]
	10239 [11]	3091 [17]	5767 [11]
F. Bit Test, Set, Reset	1674 [9]	832 [3]	891 [4]
	1542 [12]	917 [9]	887 [9]
	1212 [17]	801 [12]	1167 [12]
			1281 [11]A
G. Runge-Kutta Int.	845966 [2]	724372 [2]	696085 [2]
	1203952 [17]	665529 [3]	696049 [4]
		1012727 [17]	777846 [11]A
			874923 [17]
H. Linked List Insertion	950 [4]	1025 [13]	834 [3]
	1741 [13]	1087 [14]	1049 [13]
	1137 [14]	1210 [17]	965 [14]
I. Quicksort	7618 [5]	74278 [5]	13315 [5]
	7540 [6]	15205 [6]	9609 [6]
J. ASCII to Float-Pt.	1330 [4]	1726 [5]	2100 [3]
	2578 [5]	1512 [7]	2270 [5]
	2226 [7]	1716 [17]	1897 [17]
K. Boolean Matrix	5576 [3]	3180 [4]	2216 [6]
	5661 [6]	3905 [6]	3154 [8]
	5277 [8]	4446 [8]	3945 [17]
L. Virtual Memory Exchange	1931 [3]	2616 [4]	2539 [7]
	1934 [7]	2911 [7]	4573 [8]
	2529 [8]	4226 [8]	2643 [17]

```
                                R     Comments

(1)         LA    2,10(0,0)     4     Set R2 to 10, the length of the vectors.
(2)         LA    3,XVEC        4     Load R3 with starting address of X vector.
(3)         LA    4,YVEC        4     Load R2 with starting address of Y vector.
(4)         SDR   2,2           2     Clear floating point reg. 2.
                                      Use it to accumulate inner product.
(5)         SR    7,7           2     Clear R7
                                      Use it as index into floating point vectors.

(6) LOOP    LE    4,0(7,3)      8     Load X(i) into floating point register 4.
(7)         ME    4,0(7,4)      8     Multiply X(i) by Y(i).
(8)         ADR   2,4           2     Sum := Sum + X(i) * Y(i).
(9)         LA    7,4(0,7)      4     Increment index by 4 bytes.
(10)        BCT   2,LOOP        4     Decrement loop count and branch back if not done
                                -----
                                26    (Loop Total)
                                260   (Loop (6-10)* 10)
(11)        STO   2,SUM         12    Store double precision result in SUM.
                                ----
                                288   Grand Total
```

Table 3-1. M Measure for IBM 370 Inner Product Example

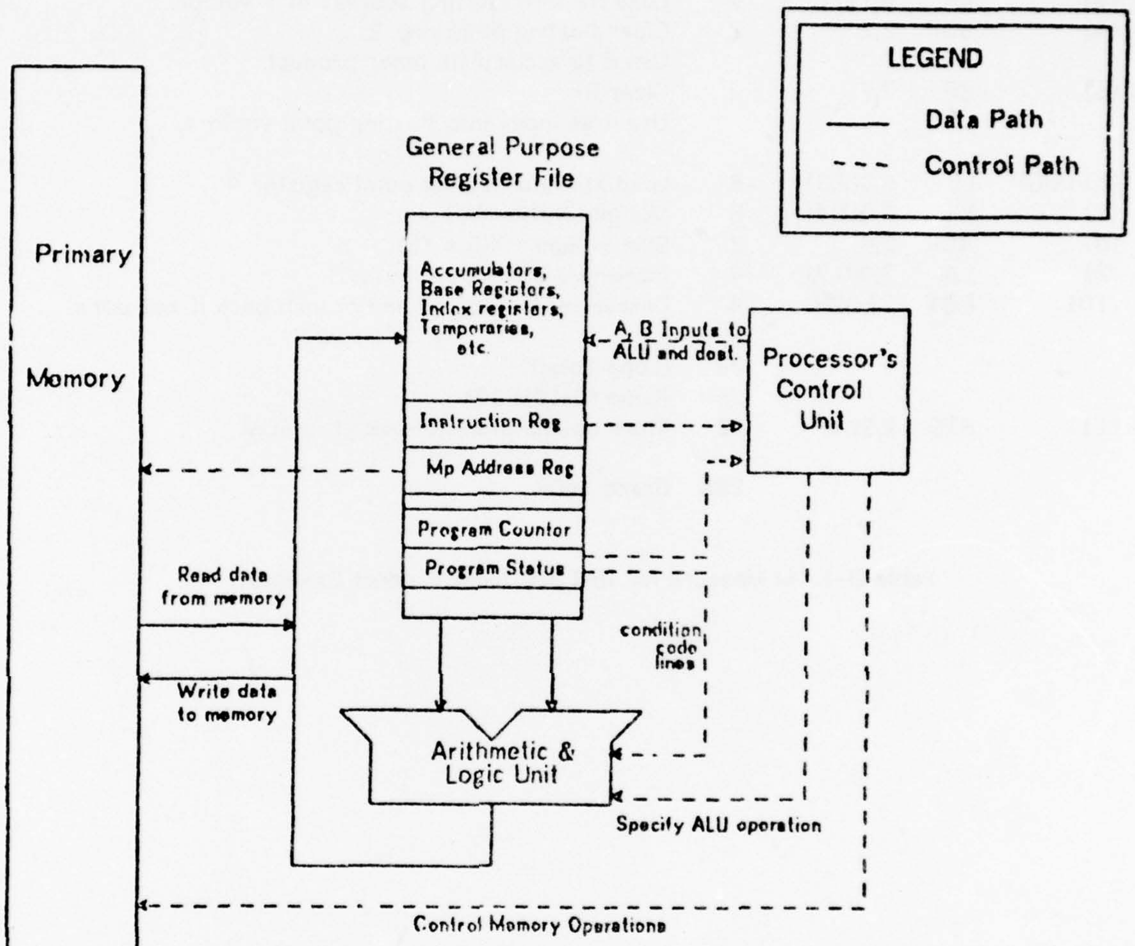


Figure 3.1: Canonical Processor Architecture

RX, RS, & SI INSTRUCTION INTERPRETATION

	R	Comment
IR<0:15> ← Mh[MAR]	2	Get halfword in instruction register
MAR ← MAR + 2	3	Incrementation counts only 1 byte
IR<15:31> ← Mh[MAR]	2	Get rest of instruction in IR
PC ← PC + 4	3	Increasing Program Counter
address interpretation	-	
instruction execution	-	
MAR ← PC	6	Set up MAR for next instruction

TOTAL	16	

RX ADDRESS CALCULATION

	R	Comment
1. B2 = 0, X2 = 0		
MAR ← IR<20:31>	5	Read 12 bits from the IR
2. B2 = 0, X2 > 0		
MAR ← IR<20:31> + R[x2]<8:31>	8	Add 12 bits from IR to 24 bits from index
3. B2 > 0, X2 = 0		
MAR ← IR<20:31> + R[B2]<8:31>	8	
4. B2 > 0, X2 > 0		
MAR ← IR<20:31> + R[B2]<8:31>	8	
MAR ← R[x2] + MAR	9	Full 24 bit (3 byte) addition

TOTAL	17	

EXAMPLE INSTRUCTION: A R4,DISP(R2,R7)

RX Add Instruction	R
RX instruction interpretation	16
address interpretation	17
MBR ← Mw[MAR]	4
R[R1] ← R[R1] + MBR	12
	--
TOTAL	49

Figure 3-2. IBM S/370 R Measure Example

		Test Program											
Phase	Programmer	A	B	C	D	E	F	G	H	I	J	K	L
I	14	all							all				
	1			all		all							
	2		all					all					
	9				all		all						
	11				all	all							
	12	all					all						
	13		all						all				
	17			all				all					
	3	370	11			832	11	11	832		832	370	370
II	4	11	832			370	832	832	370		370	11	11
	17	832	370			11	370	370	11		11	832	832
	5									all	all		
	8											all	all
III	6									all		all	
	7										all		all

Figure 4-1. Layouts of Phase I, II, and III Designs
"all" designates all three machines

Comparison of Machines \ Measure	$\sqrt{S}$	ln M	ln R
$M_3 - M_1$	-.586	.018	.012
	(-3.696,2.524)	(-.430,.466)	(-.449,.474)
$M_3 - M_2$	-3.535	-.655	-.717
	(-6.645,-.425)	(-1.103,-.207)	(-1.178,-.255)
$M_2 - M_1$	2.949	.673	.729
	(-.161,6.059)	(.225,1.121)	(.267,1.191)
$\frac{1}{2}(M_1+M_3)-M_2$	-3.242	-.664	-.723
	(-5.936,-.548)	(-1.052,-.276)	(-1.122,-.323)

model (5.1): M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-1. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase I

Measure	$\sqrt{S}$	ln S	ln M	ln R
Machine Effects				
M_1	-.788	-.148	-.230	-.247
M_2	2.161	.354	.443	.482
M_3	-1.374	-.205	-.212	-.235
μ_1		.862	.795	.781
μ_2		1.425	1.557	1.619
μ_3		.815	.809	.791

M_1, μ_1: effects for PDP-11

M_2, μ_2: effects for IBM S/370

M_3, μ_3: effects for Interdata 8/32

Table 5-2. Estimates of Machine Effects in Models (5.1) and (5.2), Phase I

Comparison of Machines \ Measure	$\sqrt{S}$	ln M	ln R
M_3-M_1	-3.806	-.295	-.348
	(-8.780,1.168)	(-1.000,.410)	(-.988,.291)
M_3-M_2	-1.585	.099	-.027
	(-6.559,3.389)	(-.606,.804)	(-.666,.613)
M_2-M_1	-2.221	-.394	-.321
	(-7.195,2.753)	(-1.099,.311)	(-.960,.318)
$\frac{1}{2}(M_1+M_3)-M_2$	.318	.247	.147
	(-3.990,4.626)	(-.364,.858)	(-.407,.701)

M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-3. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase III

Measure	$\sqrt{S}$	ln S	ln M	ln R
Machine Effects				
M_1	2.009	.133	.229	.223
M_2	-.212	.042	-.165	-.098
M_3	-1.797	-.174	-.066	-.125
μ_1		1.142	1.257	1.250
μ_2		1.043	.848	.907
μ_3		.840	.936	.882

M_1, μ_1:	effects for PDP-11
M_2, μ_2:	effects for IBM S/370
M_3, μ_3:	effects for Interdata 8/32

Table 5-4. Estimates of Machine Effects in Models (5.1) and (5.2), Phase III

Comparison of Machines \ Measure	$\sqrt{S}$ $\alpha = .67$	ln M $\alpha = .66$	ln R $\alpha = .61$
M_3-M_1	-1.649	-.088	-.128
	(-4.119,.821)	(-.442,.266)	(-.517,.261)
M_3-M_2	-2.892	-.399	-.448
	(-5.362,-.422)	(-.753,-.045)	(-.837,-.059)
M_2-M_1	1.243	.310	.320
	(-1.227,3.713)	(-.044,.664)	(-.069,.708)
$\frac{1}{2}(M_1+M_3)-M_2$	-2.067	-.354	-.384
	(-4.207,.073)	(-.661,-.047)	(-.721,-.047)

M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-5. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase I and Phase III Data Combined

Measure	$\sqrt{S}$	ln S	ln M	ln R
Machine Effects	$\alpha = .67$	$\alpha = .47$	$\alpha = .66$	$\alpha = .61$
M_1	.135	.001	.075	.064
M_2	1.378	.189	.236	.256
M_3	-1.514	-.189	-.163	-.192
μ_1		1.001	.928	.938
μ_2		1.208	1.266	1.292
μ_3		.828	.850	.825

M_1, μ_1: effects for PDP-11

M_2, μ_2: effects for IBM S/370

M_3, μ_3: effects for Interdata 8/32

Table 5- 6. Estimates of Machine Effects in Models (5.1) and (5.2), Phase I and Phase III Data Combined

Measure		$\sqrt{S}$	ln M	ln R
Sum of Squares	Degrees of freedom			
Programmers	2	.027	.018	.026
Test Programs	8	.623	.653	.660
Machines	2	.132	.076	.068
Programmers X Machines	2	.039	.053	.047
Test Programs X Machines	8	.132	.124	.121
Test Programs x Programmers	4	.047	.076	.078

Table 5-7. Phase II ANOVA Calculations
Proportion of Variance Attributable to Each Sum of Squares

ARCHITECTURE	*S*	*M*	*R*
PDP-11	1.00	0.93	0.94
IBM S/370	1.21	1.27	1.29
Interdata 8/32	0.83	0.85	0.83

Table 6-1 Average Performance of the Architectures on the 12 test Programs.

AN ARCHITECTURAL RESEARCH FACILITY: ISP DESCRIPTIONS, SIMULATION, DATA COLLECTION

Mario R. Barbacci
Carnegie-Mellon University and
Naval Research Laboratory

Daniel P. Siewiorek
Carnegie-Mellon University and
Naval Research Laboratory

Robert Gordon
Naval Underwater Systems Center

Rosemary Howbrigg
Naval Underwater Systems Center

and

Susan Zuckerman
Naval Research Laboratory

TABLE OF CONTENTS

ABSTRACT

The objectives of this paper are twofold. In the first place we discuss some issues related to the formal description of computer systems and how these issues were handled in a specific project, the selection of a standard computer architecture for the Army/Navy Computer Family Architecture (CFA) project. The second purpose is to present a methodology for automatically gathering architectural data which can be used for evaluation and comparison purposes. We will not discuss the rationale behind the selection of specific test programs and the statistical experiment set up to ascertain the influence of the programmers, the test programs, and the machine architecture on the results. These issues belong in a companion paper.

1. Introduction

There have been many attempts to specify computer architectures in some formal notation. The CFA project included, to our knowledge, the first attempt to describe the complete instruction set of several large, commercially available architectures. The candidate architectures were the IBM S/370, DEC PDP-11, and the Interdata 8/32. The experiment described in this paper involved the preparation of formal computer descriptions, the execution of machine language programs under an instrumented simulator, and the collection of data used to evaluate the architectures. Three aspects of the experiment are important to observe: 1) We did not implement specific simulators, tailored for each architecture; the system used in this project is a general purpose computer simulator driven by a formal machine description, 2) We executed a large number of test programs *, each ranging from less than a dozen

* A total of 114 simulation runs were executed. They correspond to a total of 70 different programs (some of which called for several test cases, in other instances a test case had to be divided into separated sub-cases.) The 70 programs were divided as follows: 26 for the PDP-11, 22 for each of the IBM S/370 and Interdata 8/32.

instructions to several hundred instructions, 3) We used real programs that had been executed on actual physical machines and then used to initialize the simulators.

The Naval Research Laboratory selected ISP [BelC71] as the notation to formally describe the candidate machines. This decision was based on the availability of expertise and software support at CMU and in the fact that ISP was better suited than other candidate notations for describing a computer architecture, independently of timing and other implementation issues * . This however, does not imply that ISP is free of blemishes. Some of its virtues and defects are discussed in [BarM75]. In this paper we will point out some characteristics of the notation that prevent a complete separation between architectural and implementation details.

Volume IV of the final report of the CFA committee [BarM76b] includes the ISP descriptions of the three candidate architectures and more information about the writing and debugging of ISP descriptions. It also discusses the issue of the correctness of the ISP descriptions and other matters which could not be covered in a short paper.

Section 2 presents a brief introduction to ISP through a simplified version of the IBM S/370 ISP description. Section 3 discusses the separation of architecture vs. implementation details. Section 4 describes the Architectural Research Facility. Section 5 describes the collection of architectural data from the simulation of ISP descriptions. Section 6 concludes the paper by outlining the areas in which future work could benefit from the use of the Architecture Research Facility.

* The CFA selection committee adopted the definition of architecture proposed by the designers of the IBM S/360: "The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and control, the logical design, and the physical implementation"[AmdG64].

2. A Typical ISP Description

The ISP notation was developed to formalize the information normally given in basic machine manuals and to supplement or, if possible, eventually replace the "programming reference manuals". Hence its essential requirements were readability, completeness, flexibility, and brevity.

The original notation was introduced for descriptive purposes and, in the context of a book [BelC71], certain ambigueties were permitted. For more formal uses, the notation had to be revised and a language named ISPL was developed between 1973-1975 [BarM76a]. Further developments on the notation continue at CMU, and a language tentatively named ISPS is being implemented. For the remainder of this paper we shall refer exclusively to ISPL, the dialect used in the description of the CFA architectures.

The example shown in Figure 1 is derived from the IBM S/370 ISP description. We will only present the main declarations and the instruction interpretation cycle *.

The control flow for all instructions in Figure 1 follows a well defined path. The main body of the ISP description is defined by the Run procedure which continuously performs a loop of instruction cycles (IFetch followed by IExec). After an instruction has been executed, a special section of code (INT) is executed. INT checks for the presence of exceptional conditions (errors or external interrupts) and performs the proper context switching to handle these conditions.

The instruction fetch section (IFetch) reads the first half-word of the instructions and from the first two bits (Instr<0> and Instr<1>) it computes the length of the

* In order to keep the examples within the space limitations of this paper, we have taken some minor liberties with the syntax of ISPL. These alterations should not overly confuse readers familiar with ISPL.

instruction (PSW<32:33>) and updates the program counter (PSW<40:63>). **IFetch** then proceeds to read one or two more half-words, the rest of the instruction.

The instruction execution section (**IExec**) uses the first two bits of the instruction (**Instr**<0:1>) to select an instruction-type specific section. The **RR**, **RX**, **RSSI**, and **SS** sections handle the corresponding instruction types. **RX**, **RSSI**, and **SS** begin by computing the effective address of the operand(s). After this step is completed the next 6 bits of the instruction (**Instr**<2:7>) are used to select a "routine" which describes the behavior of the instruction.

If any errors are detected during the instruction cycle (address boundary errors, illegal operations, storage protections, etc) the rest of the instruction is aborted and the proper error code is set in the PSW. This premature termination allows the interrupt handler (INT) to take care of the situation (the usual mechanism is to switch PSWs thus automatically starting the execution of interrupt specific system routines).

We have tried to keep the example as simple as possible by avoiding any details beyond those extrictly necessary to follow the example. In particular, the reader might have noticed that we were making explicit references to fields of the Instruction Register (**Instr**) and the Program Status Word (PSW). It is clear that when we deal with large descriptions such explicit references tend to become cumbersome and error prone *. The following section deals with the issues of how to improve the readability and writeability of ISP descriptions by using abstractions like pseudo-registers, procedures, temporary registers, etc.

* Even though some portions of the Architectures were left out of the ISP descriptions, notably the Floating-Point Instructions, the ISP descriptions used in this project are non-trivial computer programs. Each description takes between 30 and 40 pages of code. The size of the descriptions (1445 lines for the PDP-11, 2345 lines for the Interdata 8/32, and 2132 lines for the IBM S/370) reflects the size of the instruction set, not necessarily the complexity of the architecture.

3. Abstractions and Implementation Dependencies

ISP can be viewed as a programming language for a specific class of algorithms, i.e. Instruction Set Processors or Architectures. Ideally, a language to describe architectures should avoid the specification of any implementation details. Any components introduced beyond these are unnecessary for the programmer of the machine and might even bias the implementor working from the description. While these items must appear in a description of an implementation, the problem arises when describing a family of machines where the abstractions and/or algorithms may vary across members of the family. The rest of this section illustrates this problem.

3.1. Abstractions

An ISP description written using only the architectural components would not only be unreadable but also unwritable. Some form of abstraction is required. The following subsections demonstrate this point by introducing pseudo-registers, procedures, and temporary registers. These abstractions may or may not have a counterpart in some or all physical implementations of the ISP description.

Pseudo-Registers.- When writing an ISP description for a real machine it immediately becomes apparent that describing everything in terms of just the components of the architecture would lead to a cumbersome and unreadable description. The concept of a pseudo-register to rename a frequently used field of a register greatly relieves this problem. For example, consider the PDP-11 which has an autoincrement addressing mode. During the address computation an architecture register, pointed to by a subfield of the current instruction, must be incremented. Dealing only with components of the architecture would yield an expression like: R[M[Pc]<2:0>] ← R[M[Pc]<2:0>] + 2 where M[Pc] represents the current instruction in memory, pointed to by the program counter. Introducing the pseudo-register **Ir** (instruction register) for the current instruction would yield: R[Ir<2:8>] ← R[Ir<2:8>] + 2. We could further define a pseudo-register, **Dr** (for destination register), for the frequently used three bit subfield **Ir<2:8>**, as in: R[Dr] ← R[Dr] + 2

The pseudo-registers may suggest a register (e.g.: **Ir**) or a set of wires (e.g.: **Dr**) in some physical implementation. In reality they may have no physical correspondence at all. In any event, pseudo-registers are a useful and necessary abstraction for readable (and writable) ISP descriptions. However creating pseudo-registers for infrequently used fields or using obscure names may defeat the usefulness of this abstraction leading to reader confusion and excessive page flipping to find definitions.

Procedures.- Just as there are frequently used register fields in a machine description, there are frequently used sequences of operations. Forming these operations into procedures greatly enhances readability.

For example, consider operand fetching. Every machine has a more or less complicated effective address calculation that is performed when accessing these operands. A memory reference to a destination operand might appear as: M[**Dest**] where **Dest** is a procedure for calculating the effective address of the destination operand. Without procedures the same reference for the PDP-11 would appear as shown in Figure 2. The situation would further be aggravated if the effective address had to be processed by some form of memory management which provides for address translation and rights checking. These operations would have to be performed in the description on top of the effective address calculation. It should be noted that many minicomputers and all larger computers have some form of memory management.

Temporaries.- Occasionally readability is improved by introducing a temporary register in cases where the operands before and after the operation are required or a complex result is used repeatedly. Figure 3 shows a portion of the memory management procedures for the PDP-11.

The **Read** procedure shows the translation of a virtual address into a physical address. A temporary Memory Address Register (**Mar**) initially contains the virtual address (the result of the effective address calculation) which is then translated into a physical address in the line that reads:

```
Mar ← (PAR[Temp]<11:0> + Mar<12:6>) @ Mar<5:0> next
```

The **PAR** (Page Address Register) and **PDR** (Page Data Register) arrays contain the necessary address translation information. A bounds check is performed before the actual memory fetch from physical memory. Without the temporary variable **Mar** the **Read** procedure would be substantially complicated by having to replace every appearance of the temporary by the complex expression given above. Of course, the temporary variable may or may not have a counterpart in some implementation.

3.2. Implementation Dependencies

There are multiple examples of details that must be specified in an implementation description but do not belong in an architecture description. Typically, these are features that exhibit model dependencies. For instance, in the specification of the interrupt handling facility of a computer system, it could be the case that because of cost/performance requirements, different models must respond to simultaneous interrupts in different orders. An ISP description must by its very nature describe a specific order of interrupt trapping, thus losing a degree of freedom that one might wish to provide the machine implementors.

Figure 4 shows how the specific order in which simultaneous interrupts are fielded is build into an ISP description. Individual bits of INTVEC indicate the presence of a pending interrupt of a given priority. When only one interrupt is pending the proper context switching will take place. When more than one is pending there will be multiple context swaps and lower priority interrupts will be delayed to be processed later (the "new PSW" associated with a low priority interrupt will be stored into the "old PSW" position associated with a higher level interrupt).

It is not clear whether having to be specific about ordering of interrupts or similar events is a bad practice. Although one can claim that machine designers will be constrained in their choice of designs, the fact still remains that somebody must write the interrupt handling software, and for these programmers the order of interrupt fielding is important. This type of dilemma occurs quite often when dealing with ISP descriptions. The solution might be simply to write model-dependent ISP procedures whenever this conflict arises and then indicate in the ISP description which version of a given procedure must be implemented for a given model.

Another problem with implementation dependencies is that the definition of the input/output behavior of an instruction might actually imply a particular implementation. For example, consider the PDP-11 Subtract instruction. The carry condition code (C) is set according to the borrow during the subtraction. The PDP-11 Processor Handbooks describes the setting of the C bit as:

"C condition code is cleared if there was a carry from the most significant bit of the result, set otherwise."

This definition implicitly assumes that subtraction is implemented by forming the two's complement and adding. Figure 5 illustrates the situation. Consider four-bit numbers and the two methods to perform subtractions, by using a subtractor, and by using an adder after forming the two's complement.

In the adder case, the carry is the complement of the borrow which is exactly the definition given by the PDP-11 Processor Handbook. The ISP description of the setting of C becomes:

```
C ← (dest - source)<16>;                                 ! Subtraction
C ← NOT (dest + NOT(source) + 1)<16>;                    ! Addition
```

As in the previous example (the order of interrupt handling), a complete algorithm had to be given. In this case, the subtractor/borrow algorithm is preferred since it presupposes only the properties of the two's complement number system. However, if an alternate implementation (such as forming the two's complement and adding) is utilized, then the implementor should be aware of possible changes in other algorithms in the ISP description.

4. The Architecture Research Facility

The facility used for the data collection phase of the CFA project is depicted in Figure 6. Reference [BarM76a] explains in full detail the features of the ISP compiler and simulator. Some familiarity with their capabilities is needed in order to understand the data collection phase described later. The following paragraphs attempt to satisfy this need.

The ISP compiler produces code for a hypothetical machine, dubbed the Register Transfer Machine (RTM). The "object code" produced by the compiler can be linked together with a program which is capable of interpreting RTM instructions. This separation between the ISP description, the RTM code, and the RTM interpreter allows the simulation of arbitrary, user defined architectures. The result of linking the RTM code with the RTM interpreter is a running program, a simulator.

The simulator accepts commands from a teletype or user designated command file. The state of the simulator can be dumped to a command file which can be read at a future date when the simulation is continued. Command files can also be used to load programs and data into the simulated target machine memory and registers.

4.1. Debugging

Most of the test programs were debugged and run on the real machines, other programs were executed exclusively under the simulator. The latter included those programs using privileged instructions that were not directly available to non-system programmers (e.g. interrupt and I/O handlers.) Results from the actual runs, whenever available, were used to check the simulated execution.

Only minor modifications and corrections were performed during the data collection phase. The largest unforeseen problem was presented by the memory management feature of the PDP-11 which was based on the PDP-11/40. The test programs which made use of this feature had been tested on a PDP-11/45 which uses different Unibus addresses for the memory management registers. This difference required minor modifications in the test programs. Most other problems were of a simpler nature and required only a few minutes to correct. It should be noted here that the simulator facility was also used to debug some programs for the Interdata 8/32 before they were executed on the real machine. This was dictated by the fact that no 8/32 was available near CMU and a large turn-around time (several days) would have complicated the debugging of the test programs.

4.2. Preparation of Simulation Tests

The ISP simulator provides commands for the loading and initialization of the simulated machine memory and internal registers. The single most important feature of the command language which permitted the fast execution and collection of statistics was the ability to read command files containing the test programs to be executed. The command language cannot handle programs in symbolic form (assembly language); it requires the preassembly of the programs into absolute, numeric, code. To get around this problem, a set of utilities was developed at CMU which permitted the transformation of assembly listings prepared by the real machine's assembler into simulation command files. This operation was performed off-line as shown in Figure 6.

Figures 7 and 8 show the transcript of a typical session using the ISP simulator. The session consists of running one of the test programs (Bit Test, Set, and Reset) on the PDP-11. The input for a simulation session consists of several files prepared off-line. These files include: The test program (derived from the assembly listing), a driver (simulation commands used to initialize the parameters for the test program), and finally, a command file with a list of those ISP procedures which must be "opaqued" (these are the procedures during which the activity counters are disabled). A typical command file, derived from an assembler listing is shown in Figure 9. This was the test program used in the sample simulator session shown in Figures 7 and 8.

4.3. Instrumentation

The ISP simulator permits the instrumentation of an ISP description by associating activity counters with each of the machine registers and memories. These counters allow the collection of statistics indicating the number of times each component of the machine is read from or written into. A separate counter is kept for each label in the ISP description. Labels are included in the ISP descriptions to identify machine instructions, addressing modes, loops (used to describe vector-like instructions like move character on the S/370), as well as other ISP procedures.

During the execution of the test programs, a data base was created by collecting dumps of the counters after each test case was completed. The files containing the counters were then processed by other, off-line, programs in order to arrive at the M and R measures.

4.4. Artificial Labels in the ISP Descriptions

Certain modifications not normally needed were made to the ISP descriptions to aid in the collection of data during the running of the test programs for the CFA project. Several labels and "do-nothing" procedures were added to identify certain phases in the instruction interpretation algorithm and to measure selected events (e.g., different addressing modes). The labels added to count these events are clearly not part of the architecture or even the implementation.

Figure 10 shows an example extracted from the S/370 ISP Description. It shows the use of artificial labels to identify different addressing modes for the RX instruction set. According to the definition of the S/360 and S/370 architectures, The RX instructions can specify both a base and an index register to be added together with the displacement field of the instruction to compute the address of the memory operand. The architecture further specifies that R[0], when specified as either a base or index register does not take place in the effective address calculation, i.e., R[0] should be specified whenever one of these two components (base or index) is missing. In the above example four dummy in-line procedures where introduced to count the number of times each possible combination of base/index modes occurs. Thus RX0000 is "executed" whenever R[0] is specified as both the base and the index register. RX00X2 is "executed" whenever R[0] is used as the base register and any of R[1:15] is used as the Index register. RXB100 is "executed" whenever R[0] is specified as the index register and any of R[1:15] is specified as the base register. Finally, **RXB1X2** is "executed" whenever R[0] is not specified as either the base or index registers. NOP is a dummy procedure which does not have any side effects.

5. Architecture Parameters

As a means of comparing architectures, three measures were defined for the CFA project [FulS77a]:

Measure of Space

S	The number of bytes used to represent a test program.

Measures of Execution Time

M	The number of bytes transferred between primary memory and the processor during the execution of the test programs.
R	The number of bytes transferred among internal registers of the processor during execution of the test program.

The S measure is a static parameter which can be computed independently of the ISP description. For the purposes of this paper we will restrict the discussion to the other two measures. The actual computation of the M and R measures was done through a semiautomatic process. The raw data collected from the simulator was used to count frequencies of instructions and addressing modes. These counters were multiplied by certain hand calculated factors in order to arrive at the M and R measures for each test program. Ideally, the ISP simulator should perform the entire operation and this would be a better approach, less subject to human errors. We had to use the hand computed factors due to our inability to determine the influence of the ISP writing style on the architecture parameters as defined above.

The exact methodology for writing ISP descriptions so that the M and R measures can be calculated automatically has yet to be developed. It is clear, however, that a careful control of the counting mechanism is paramount to the collection of meaningful data. During the data collection phase we made use of the following techniques towards this goal.

Opaqued Procedures.- A Simulator command allows the selective masking of in-line and off-line procedures. Masking or opaquing a procedure inhibits all activity counts inside the body of the procedure.

Certain operations, such as incrementing the program counter after an instruction, or the setting of the condition codes as a result of an instruction do not affect the R measure and should not be counted. This is typical of those actions which, in a reasonable implementation, would be done using ad-hoc circuitry, separate from the main operational units of the machine. These operations could be implemented by combinational logic (e.g.: setting condition codes from ALU lines), special registers (e.g.: using a counter instead of a simple register for the program counter), or even complex sequential networks (e.g.: the virtual address translation can be performed using its own arithmetic units and data paths).

Operations like those described above can be easily marked by adding artificial labels to the ISP description and then disabling the counters while the selected operation is being performed.

Pseudo-Register Chains.- Every component declared in an ISP description has activity counters associated with it. When a register is defined in terms of another register, such as: Pc<15:0> := R[7]<15:0>; a redefinition chain is established. Accesses higher up in the chain increment all counters lower in the chain but not vice-versa. In the above example an access of the Pc causes the register file counter for R to be incremented but accessing R[7] does not increment the program counter (Pc). By establishing appropriate redefinition chains, distinction between access types can be maintained. One variation of this technique is the use of "shadow" registers. For example two instruction registers can be defined: **Ir<15:0> := Irl<15:0>;** where **Irl** is the shadow register. The loading of the Ir from memory is to be counted in the R measure, however, the combinational logic decoding of the instruction and effective addressing mode is not to be counted. The former is performed on Ir, the latter on Irl thus distinguishing the two different types of accesses.

<u>Memory Access Procedures</u>.- Modern machines provide the user with an address space defined in terms of small units of information, typically 8-bit bytes. For convenience, however, the architectures also define larger access units in multiples of bytes. Thus, the IBM S/370 provides bytes, half-words, full-words, and double-words. Since the physical memory is the same, the ISP description must declare the different address spaces by building a redefinition chain in which the different address spaces are declared as "pseudo-memories" so that the M measure component of each address space is properly accounted for.

Machines like the PDP-11 add some more complexity to the issue of having multiple address spaces. The PDP-11 architecture defines the concept of an I/O page as a reserved portion of the address space, not necessarily implemented as a physical memory. Addresses in the upper 4K bytes of the PDP-11 are used to address I/O devices, machine registers, etc. Addresses in the I/O page must be handled differently when computing the M measure. If one attempts to include in-line address checks in the ISP description, the description quickly becomes bulky and unreadable. A satisfactory solution is simply to define memory access procedures (Read and Write), which can then be properly instrumented, thus enabling the automatic computation of the M measure.

Temporary Registers.- The automatic computation of the R measure is more difficult. In an ISP description there are three types of registers to consider: architectural, standard implementation, and temporaries. Architectural registers and certain standard implementation registers (instruction register, memory address register, and memory buffer register) can be handled using the same techniques used to automate the M measure (declaration chains and encapsulating procedures). Handling temporary registers presents a more difficult problem. The number, type, and manipulation of temporary registers are a matter of writing style.

Architecture parameters which are based solely on architecture registers while ignoring temporary registers introduced for clarity might overlook hidden computations performed on these registers. Unlike the memory, architectural registers, and standard implementation registers, a tightly defined writing style cannot be developed for temporary registers. One solution would be to use well known expression optimization techniques [WulW75] on the ISP description to uniformly minimize the temporary register activity. Hopefully the optimization would lead to similar results for equivalent algorithms.

Architectural parameters should be independent of the experience, style, and objectives of the ISP writer. This will then guarantee that the ISP descriptions which make use of abstractions (pseudo-registers, procedures, and temporary registers, etc) to enhance clarity and readability will not be penalized. By the same token, no advantage should be derived from the use of "clever" programming tricks which might attempt to bias the measurements.

6. Advantages of an Architectural Research Facility

Although for the purposes of this paper we have presented the uses of the ISPL compiler and simulator in the context of a specific project, we should point out the wider range of applications in which a system like ARF can be of great value.

6.1. A Simulator as a Training Tool

In this paper we described how machine language test programs can be executed under the simulator. The implied assumption during the data collection phase was that we were dealing with correct, finished programs. With no extra effort the ISP simulator can be a powerful training device for novice programmers. Speed of simulation is not an issue in this application. Programmers learning a new machine language tend to spend long hours single-stepping via the machine console. An interactive simulator can easily satisfy the needs of these users, while providing much better diagnostic and debugging facilities than a computer console (did you ever see a "help" button on a machine?.) ISP descriptions exist for the following machines: DEC PDP-8, PDP-10, PDP-11, IBM S/370, Interdata 8/32, and Intel 8080.

6.2. Architecture Evaluation

The S, M, and R *measures are* by *no means* the only set of architecture parameters one might wish to evaluate. Nothing in the ISP simulator depends upon this particular set of parameters. The instrumentation in the simulator allows counting every event we care to define by simply labelling the event. There is no need to create new procedures which might impact the organization or readability of the description; even a single register transfer operation can be labelled and counted.

6.3. Experimentation

Once the initial effort of writing an ISP description is accomplished, only moderate effort is required to perturb it to reflect proposed or actual changes in the architecture. Thus the effect of a modification in an architecture can be measured and studied before any funds are commited to the development of a new machine. By a careful design of the ISP description it is possible to pattern a description along the lines of the organization of the physical machine. Thus one would be able to measure and evaluate different models of the architecture. For instance, functional units and data paths can be represented by separate procedures in the ISP description. An ISP description could then be parameterized to invoke these procedures in different order, concurrently or sequentially, with or without intermediate steps, etc. as the different models differ in their implementation. An example might be determining the effect of a cache memory on the apparent instruction execution speed in high performance implementations.

6.4. Machine Relative Software

As the number of different architectures coming into existence increases every year, it is becoming more and more expensive to develop the necessary software support base that allows the effective use of these machines. The availability of user micro-programmable machines enlarges the space of possible architectures to the point that automatic software generation systems will become a necessity. Tools that operate relative to a computer description could represent a significant breakthrough in the manner that computer systems (hardware/software) are designed and evaluated. The Advanced Research Projects Agency (ARPA) of the Department of Defense is currently sponsoring this area of research at CMU and elsewhere [BarM74].

In the future one can foresee hardware and software automation systems that take as input computer descriptions, and language and problem specifications; and from these, generate operating systems, compilers, and other support and application software automatically. Other areas of current research include automatic diagnostic generation, microcode generation, machine verification, etc.

Formal computer descriptions will play an increasing and important role in the evaluation, procurement, verification, and programming of computers. The ARF facility is a step in this direction.

```
S370:=
begin  declare
       Memory[0:"FFFFFF]<0:7>;                                          ! Primary Memory
       R[0:15]<0:31>;                                         ! General Purpose Registers
       PSW<0:63>;                                                  ! Program Status Word
       . . . . . . .                             ! Auxiliary Registers (Instr, Mar, Mbr, etc.)
       eralced                                                   ! End of Declarations

Run:= begin                                                   ! Main Executable Program
       IFetch:= begin                                        ! Instruction Fetch Section
              Mar←PSW<40:63> next                         ! Initial Instruction Address
              Instr<0:15>←Memory[Mar:Mar+1] next    ! Read First Half-Word of Instruction
              PSW<32:33>←Instr<0>+Instr<1>+1 next                ! Instruction Length
              PSW<40:63>←PSW<40:63>+PSW<32:33>*2 next              ! Program Counter
              . . . . . . .                             ! Fetch the rest of the Instruction
              end;
       IExec:= begin                                     ! Instruction Execution Section
              decode Instr<0:1> =>                             ! Select Instruction Type;
              RR:=    begin                                 ! RR Instruction Decode Table
                      (decode Instr<2:7> => . . . . . )         ! Select RR Instructions
                      end;
              RX:=    begin                                 ! RX Instruction Decode Table
                      Mar←Instr<20:31> next                            ! Displacement
                      (if Instr<16:19> => Mar←Mar+R[Instr<16:19>]) next        ! Base
                      (if Instr<12:15> => Mar←Mar+R[Instr<12:15>]) next       ! Index
                      (decode Instr<2:7> => . . . . . )         ! Select RX Instructions
                      end;
              RSSI:=  begin          ! RS,SI Instruction Decode Table
                      Mar ← Instr<20:31> next                          ! Displacement
                      (if Instr<16:19> => Mar ← Mar+R[Instr<16:19>]) next      ! Base
                      (decode Instr<2:7> => . . . . . )      ! Select RS, SI Instructions
                      end;
              SS:=    begin                                 ! SS Instruction Decode Table
                      AMar1←Instr<20:31>; AMar2←Instr<36:47> next     ! Displacements
                      (if Instr<16:19> => AMar1←AMar1+R[Instr<16:19>]);        ! Base
                      (if Instr<32:35> => AMar2←AMar2+R[Instr<32:35>]) next    ! Base
                      (decode Instr<2:7> => . . . . . )         ! Select SS Instructions
                      end;
              end;
       INT:=  begin . . . . . end next                      ! Interrupt Handling Section
       Run                                                     ! Repeat Main Procedure
       end
end
```

Figure 1 - A Simplified Version of the IBM S/370 ISP Description

```
M[    decode Dd =>
          (decode Dm =>                                        ! Direct Addressing
               #37400@Dr;                                        ! Register Mode
               R[Dr]←R[Dr]+2 next R[Dr]-2;                  ! Autoincrement Mode
               R[Dr]←R[Dr]-2 next R[Dr];                    ! Autodecrement Mode
               M[Pc+2] + R[Dr]                                     ! Index mode
               );
          (decode Dm =>                                            ! Deferred Mode
               M[#37400@Dr];                                       ! Register mode
               R[Dr]←R[Dr]+2 next M[R[Dr]-2];               ! Autoincrement Mode
               R[Dr]←R[Dr]-2 next M[R[Dr]];                 ! Autodecrement mode
               M[M[Pc+2] + R[Dr]]                                  ! Index mode
               )
      ]
```

Figure 2 - Inline Effective Address Calculation

```
Read:=begin
       Temp ← Mar<15:13> next
       Mar ← (PAR[Temp]<11:8> + Mar<12:6>) @ Mar<5:8> next     ! Compute Physical Address
       (if not PDR[Temp]<2:1> => Abort) next
       (if (Mar<12:6> gtr PDR[Temp]<14:8>) and not PDR[Temp]<3> => Abort) next
       (if (Mar<12:6> lss PDR[Temp]<14:8>) and PDR[Temp]<3> => Abort) next
       . . . . . . .                                             ! Read from Physical Memory
       end;
```

Figure 3 - A Portion of the PDP-11 Memory Management

```
Int:=   begin
        Temp←PSW<32:33> next                          ! Save Instruction Length
        (if INTVEC<0> AND PSW<13> =>               ! Handle Priority (1) Interrupts
                . . . . . . .
                ) next
        (if INTVEC<1> =>                           ! Handle Priority (2) Interrrupts
                . . . . . . .
                ) next
        (if INTVEC<2> =>
                . . . . . . .
                ) next
        (if INTVEC<3> AND PSW<0:7> =>              ! Handle Priority (3) Interrupts
                . . . . . . .
                ) next
        (if INTVEC<4> AND IOMSK =>                 ! Handle Priority (4) Interrupts
                . . . . . . .
                ) next
        PSW<16:31>←0; PSW<32:33>←Temp ! Reset Instruction Length & Interrupt Code
        end;
```

Figure 4 - Explicit Interrupt Processing Order in the IBM S/370

```
5 - 3 = 2 (no borrow)   3 - 5 = -2 (borrow)

   0101                    0011      Subtracting
   0011                    0101
   ----                    ----
0 0010                  1 1110
borrow                  borrow

   0101                    0011      Adding Two's Complement
   1101                    1011
   ----                    ----
1 0010                  0 1110
carry                   carry
```

Figure 5 - Implementation Dependant Condition Code Setting

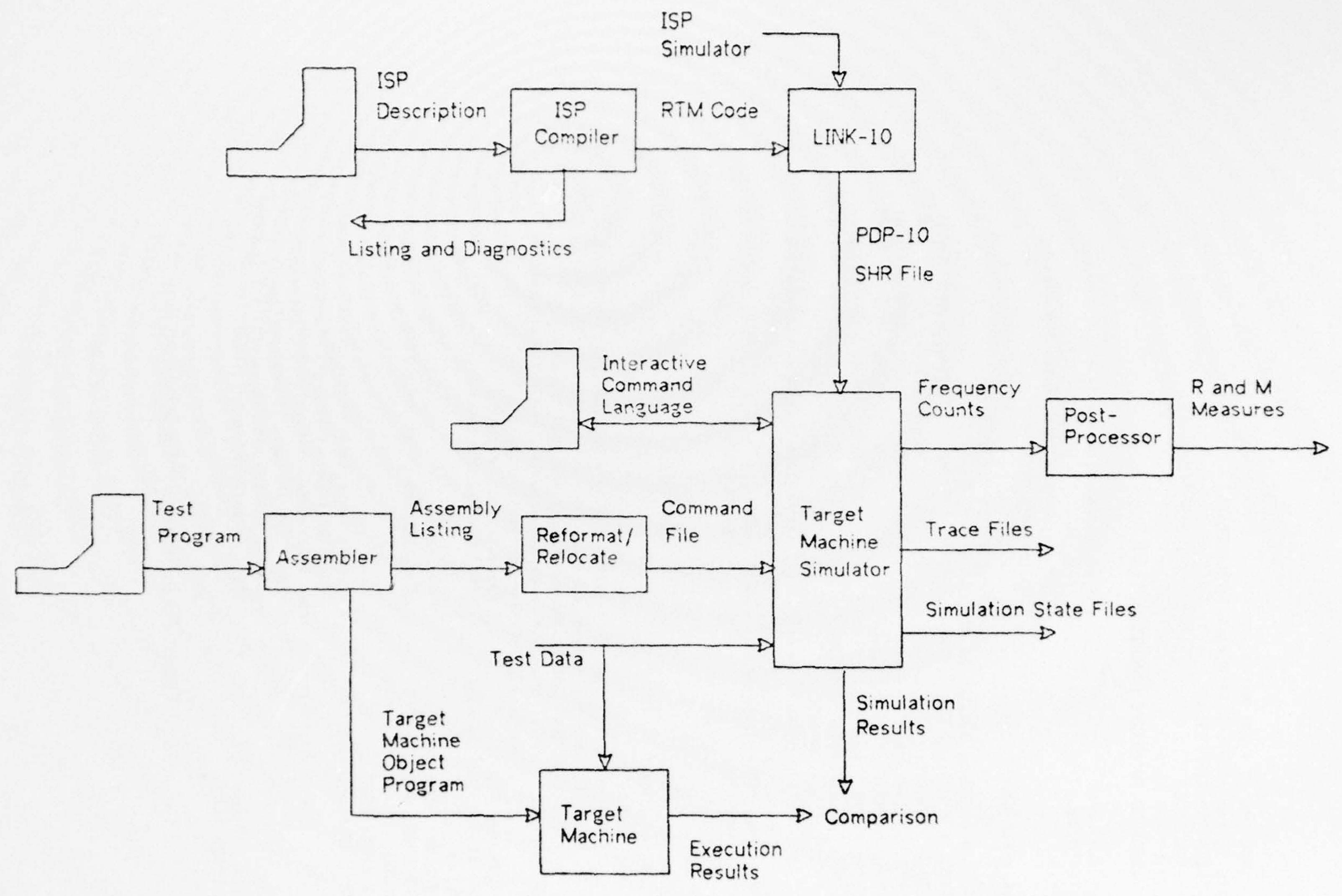


Figure 6 - Test Program Execution Under ARF

```
ru pdp11m
ISP SIMULATOR V3 - NRL ARF STAGE 2
Friday 10 Sep 76 17:13:50 PDP11M.ISP[L410MB25]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>read fadl.sim                   ! Read in the benchmark file
>>RADIX OCTAL
>>DECHO                          ! The benchmark file disables the listing
                                 ! on the user terminal.
>>100 LINES READ
>read fa.dr3                     ! Read in the driver file
>>!     HERE COMES THE DRIVER (CALLS)
>>SETVAL MW[3000]←013746 005202          !       MOV     @#5202,-(SP)    ; F
>>SETVAL MW[3002]←013746 005204          !       MOV     @#5204,-(SP)    ; N
>>SETVAL MW[3004]←012746 004000          !       MOV     #4000,-(SP)     ; A1
>>SETVAL MW[3006]←012746 005200          !       MOV     #5200,-(SP)     ; RC
>>SETVAL MW[3010]←012746 005206          !       MOV     #5206,-(SP)     ; W
>>SETVAL MW[3012]←004737 001000          !       JSR     PC,@#1000       ; BTSR
>>SETVAL MW[3014]←000000                 !       HLT
>>      ! The above sequence of PDP-11 instructions pushes the parameters
        ! onto the stack, call the benchmark as a routine, and halt.
>>SETVAL MW[2000]←123457 071234 167006 145670   !       BIT STRING
>>SETVAL MW[2500]←0              !       RETURN CODE
>>SETVAL MW[2501]←2              !       F
>>SETVAL MW[2502]←25             !       N
>>SETVAL MW[2503]←0              !       WORK AREA
>>SETVAL PC←6000
>>SETVAL SP←200
        ! The above sequence initializes the data (parameters), the stack
        ! pointer and the program counter (which now points to the code
        ! sequence that pushes the parameters and call the routine.
>>SETVAL A←0    ! This is an ISP internal variable - indicates whether the
                ! machine is running, halted, or waiting.
>>SETCTR ALL 0,0                 ! Reset activity counters
>>READ OPQ11.SIM[L410MB25]       ! PDP11 Opaqued Procedures
>>>DECHO
>>>53 LINES READ
>>READ UUO11.SIM[L410MB25]       ! UNIMPLEMENTED OPERATION BREAKS
>>>DECHO
>>>15 LINES READ
>>TRACE IR,PC,R,MWIO             ! Trace a few selected registers
                                 ! IR is the Instruction Register,
                                 ! PC is the Program Counter (R[7]),
                                 ! R[0:7] are the general registers,
                                 ! MWIO is the I/O page (R is mapped onto MWIO)
>>BREAK JSR,RTS                  ! Break on selected instructions
>>26 LINES READ
```

Figure 7 - Initialization of a Simulation Run

```
>start inter                             ! Here we start the simulation
@ INTER  + 15   IR      = 13746
@ INTER  + 20   PC      = 6002
@ SINCD  + 22   R       [ 7]= 6004
@ DDECRD + 21   R       [ 6]= 176
@ INTER  + 15   IR      = 13746

  . . . . . . . .        ! Pushing Parameters

@ INTER  + 15   IR      = 12746
@ INTER  + 20   PC      = 6022
@ SINCD  + 22   R       [ 7]= 6024
@ DDECRD + 21   R       [ 6]= 166
@ INTER  + 15   IR      = 4737
@ INTER  + 20   PC      = 6026
BREAK AFTER JSR                          ! The simulation stops on a breakpoint
*setctr all 0,0                          ! The real benchmark starts here, we must
                                         ! reset all counters (they were modified
                                         ! during the benchmark calling sequence)
*cont                                    ! we continue the simulation
@ DINCRD + 22   R       [ 7]= 6030
@ JSR    + 14   R       [ 7]= 6030
@ JSR    + 15   PC      = 1000
@ INTER  + 15   IR      = 10046
@ INTER  + 20   PC      = 1002

  . . . . . . . . ! Program Execution Trace

@ INTER  + 20   PC      = 1072
@ SINCD  + 22   R       [ 6]= 164
@ WRITE  + 131  MWIO    [ 374000]= 0
@ INTER  + 15   IR      = 207
@ INTER  + 20   PC      = 1074
BREAK AFTER RTS                          ! the simulation stops at the end of the
                                         ! benchmark (the return instruction)

*outctr fadl.rm3                         ! we dump all the counters into a file
*cont                                    ! we continue the simulation
@ RTS    + 2    PC      = 1074
@ RTS    + 7    R       [ 7]= 6030
@ INTER  + 15   IR      = 0
@ INTER  + 20   PC      = 6032
SIMULATION COMPLETED                     ! we executed the Halt instruction
RUN TIME(10 usec units)=831678
RTM OPS EXECUTED=4535
>exit                                    ! we finish the session
EXIT
```

Figure 8 - Program Execution Trace

```
RADIX OCTAL
DECHO
!CFAF   MACN11   V003F   5-JUL-76   12:54   PAGE 1
!BTSR1  M11
!
 . . . . . . . . ! Program, Programmer Identification (Supressed)

!     13                                      01300   ; Offsets of parameters from stack p
!     14                                      01400   ;
!     15              000004                  01500   SAVE=4              ; we need to save 2
!     16                                      01600   ;
!     17              000016                  01700   F=12+SAVE           ; function code
!     18              000014                  01800   N=10+SAVE           ; relative bit numbe
!     19              000012                  01900   A1=6+SAVE           ; address of bit str
!     20              000010                  02000   RC=4+SAVE           ; address of return
!     21              000006                  02100   WORK=2+SAVE         ; address of work ar
!     22                                      02200   ;
!     23        000000'                       02300   BTSR:
!     24        000000' 010046                02400           MOV     R0,-(SP)
!     25        000002' 010146                02500           MOV     R1,-(SP)
!     26        000004' 005076  000010        02600           CLR     @RC(SP)         ; ze
!     27        000010' 016600  000014        02700           MOV     N(SP),R0        ; ge

 . . . . . . . . ! Relocatable Object Code Listing

!     41        000066' 012601                04100   QUIT:   MOV     (SP)+,R1        ; ex
!     42        000070' 012600                04200           MOV     (SP)+,R0
!     43        000072' 000207                04300           RTS     PC
!     44        000074' 150110                04400   SET:    BISB    R1,@R0          ; FC
!     45        000076' 000773                04500           BR      QUIT
!     46              000001                  04600           .END

 . . . . . . . . ! Cross-Reference Listing

!
                              ! Here begin the simulation commands
                              ! derived from the above listing
                              ! relocation address = word 400 (octal) = byte 1000
!
SETVAL MW[400]←010046
SETVAL MW[401]←010146
SETVAL MW[402]←005076 000010
SETVAL MW[404]←016600 000014

 . . . . . . . . ! Target Machine Program Loading

SETVAL MW[433]←012601
SETVAL MW[434]←012600
SETVAL MW[435]←000207
SETVAL MW[436]←150110
SETVAL MW[437]←000773

ECHO
```

Figure 9 - A Command File Derived from an Assembly Listing

```
RX:=    begin
        Mar←Instr<28:31> next
        (decode (Instr<16:19> NEQ 0)@(Instr<12:15> NEQ 0)=>
        \00     RX0000:=        (NOP);                          ! No Base, No Index
        \01     RX00X2:=        (NOP);                          ! No Base, Indexing
        \10     RXB100:=        (NOP);                          ! Base, No Index
        \11     RXB1X2:=        (NOP)                           ! Base, Indexing
                ) next
        (if Instr<16:19> => Mar←Mar+R[Instr<16:19>]) next
        (if Instr<12:15> => Mar←Mar+R[Instr<12:15>]) next
        (decode Instr<2:7> =>
        . . . . . .                                             ! Select RX Instructions
                )
        end;
```

Figure 10 - Use of Artificial Labels

References

[AmdG64] Amdahl, G. M., Blaauw, G. A., and Brooks, F. P., "Architecture of the IBM System/360", IBM Journal of Research and Development, Vol. 8, No. 2, April 1964, pp. 87-101.

[AndV74] Anderson, V. L. and McLean, R. A., Design of Experiments, a Realistic Approach, Marcel Dekker, Inc., New York, 1974.

[BarM74] Barbacci, M.R. and Siewiorek D.P.: Some Aspects of the Symbolic Manipulation of Computer Descriptions. Department of Computer Science, Carnegie-Mellon University, July 1974.

[BarM75] Barbacci, M.R.: "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems". IEEE Transactions on Computers, Volume C-24, Number 2, February 1975, pp. 137-149.

[BarM76a] Barbacci, M.R.: "The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator". Technical Report, Department of Computer Science, Carnegie-Mellon University, 1976.

[BarM76b] Barbacci, M.R, D.P. Siewiorek, R. Gordon, R. Howbrigg, and S. Zuckerman: "Architecture Research Facility: ISP Descriptions, Simulation, Data Collection." Volume IV of Computer Family Architecture Selection Committee Final Report. Naval Research Laboratory, Washington D.C., December 1976.

[BelC71] Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971.

[BerN75] Bernwell, N. (editor), Benchmarking: Computer Evaluation and Measurement, John Wiley & Sons, New York, 1975.

[BoxG64] Box, G. E. P. and Cox, B. R., "An Analysis of Transformations", The Journal of the Royal Statistical Society, Series B, Vol. 26 (1964), 211-252.

[GMLC75] Computer Review (formerly Computer Characteristics Review, GML Corporation, Lexington, MA, 02173, 1975.

[ConW59] Connor, W. S. and Zelen, M., "Fractional Factorial Experiment Designs for Factors at Three Levels", National Bureau of Standards, Applied Mathematics Series Vol. 54, 1959.

[CorJ77] Cornyn, J.J, Smith, W.R., Svirsky, W.R., and Coleman, A.H.: "Two Life-Cycle Cost Models for Comparing Computer Architectures". Submitted to National Computer Conference, NCC-77.

[DavO71] Davies, O. L. (editor), Design and Analysis of Industrial Experiments, 2nd ed., Oliver and Boyd, Edinburgh, 1971.

[FulS76a] Fuller, S. H., Stone, H. S., and Burr, W. E., "Selection of Candidate Computer Architectures and Initial Screening." Volume II of Computer Family Architecture Selection Committee Final Report, Naval Research Laboratory, Washington, D.C. 20375. 1 December, 1976.

[FulS76b] Fuller, S.F., W.E. Burr, P. Shaman, and D. Lamb: "Evaluation of Computer Architectures via Test Programs". Volume III of Computer Family Architecture Selection Committee Final Report. Naval Research Laboratory, Washington D.C., 1 December 1976.

[FulS77a] Fuller, S. H., Burr, W. E., Shaman, P., and Lamb, D. A., "Evaluation of Computer Architectures via Test Programs." This Volume.

[FulS77b] Fuller, S.F., H.S. Stone, and W.E. Burr: "Initial Selection and Screening of the CFA Candidate Computer Architecture." This Volume.

[LucH71] Lucas, H. C., "Performance Evaluation and Monitoring", ACM Computing Surveys, 3, 3 (1971), pp 79-91.

[PopG74] Popek, G. J., and Goldberg, R. P., "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, Vol. 17, No. 7, July 1974, 412-421.

[RaoC73] Rao, C. R., Linear Statistical Inference and its Applications, 2nd ed., John Wiley & Sons, New York, 1973.

[SmiW76] Smith, W.R., J.J. Cornyn, A.H. Coleman, W. Svirsky, R. Estell, P. Sabin: "Life Cycle Cost Models for Comparing Computer Family Architectures". Submitted to National Computer Conference, NCC-77.

[StoH75] Stone, H. S. (editor), Introduction to Computer Architecture, Science Research Associates, Chicago, 1975.

[StoH76] Stone, H. S., "An Audit of the Selection Criteria for Computer Family Architecture," CFA memorandum, January, 1976. Distributed at the 18-20 February CFA meeting.

[WagJ76] Wagner, J., B. Lieblain, J. Rodriguez, H.S. Stone: "Evaluation of the Candidate Architectures for the Military Camputer Family". Submitted to National Computer Conference, NCC-77.

[WicB73] Wichmann, B. A., Algol 60 Compilation and Assesment, Anderson Press, New York, 1973.

[WulW75] Wulf, W. et. al.: The Design of an Optimizing Compiler. American Elsevier, Programming Language Series, New York, 1975.